Unit-3
Queue

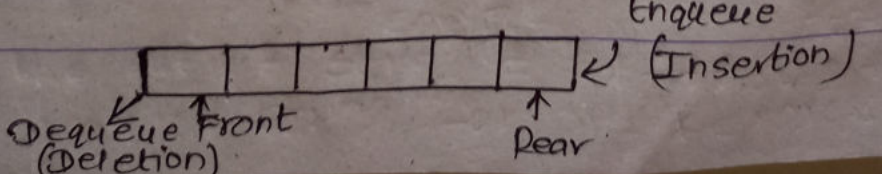
#Queue:

A queue is a non-primitive linear data structure in which operation like insertion of an element is allowed from one-end of the queue known as rear end and the deletion of an element is allowed from other end known as front end. The element in the queue is processed in the same order as it receives so queue is also referred as FIFO (First In First Out).

The insertion operation is referred to as 'Enqueue' and deletion operation is referred as 'Dequeue'.

Queue uses two veriables rear and front. Rear is incremented while inserting an element into the queue and front is incremented while deleting an element from the queue. The condition when front is NULL indicates that the queue is empty. The condition when rear is 'n' indicates overflow (n = size of queue).

Consider an example of railway reservation counter, where the people come to the counter from one end (rear end) and goes from another end (front end) after getting reservation.



Applications of Queue:

(1) When jobs are submitted to the printer, they are arranged in order of arrival. Thus, job sent to a line printer are placed on a queue.

(2) Virtually every real life line is supposed to be queue. For e.g. lines at ticket counter.

(3) Calls to a large company are generally placed on queue when operator are busy.

(4) Time sharing system for use of CPU.

(5) Queue is used to maintain the playlist in media player in order to add and remove the songs from the playlist.

(6) Queues are used in operating system for handling interrupts.

Primitive (Basic) operations on queue:

(1) Enqueue (q, x):

To insert an item 'x' at the rear of queue or add an element to the queue from the rear end.

(2) Dequeue (q):
Remove an element from queue from front end.

(3) Front or peek():
Return the object or element that is at the front of the queue without removing

(4) Empty:
Return true if the queue is empty otherwise return false.

(5) Size:
Returns the number of items in the queue.

(6) Display:
Retrieve all elements of queue.

Queue as an ADT:
A queue 'q' of type 'T' is a finite sequence of elements with the operations.

(1) Make Empty (q): To make queue as an empty.

(2) IsEmpty (q): To check whether the queue 'q' is empty or not. Return true if queue is true otherwise return false.

(3) Isfull (q):

To check whether the queue 'q' is full or not. Return true if queue is full otherwise return false.

(4) Enqueue (q, x):

To insert an item 'x' at the rear of the queue if and only if queue is not full.

(5) Dequeue (q):

To delete an item from the front of the queue if and only if queue is not empty.

(6) Traverse (q):

To read entire queue that is display the content of queue.

Q. (i) If $a < b$, 0

(ii) If $b <= a$, $\varphi(a,b) = \varphi(a-b, b) + 1$

Find the value of $\varphi(2,3)$ and $\varphi(4,3)$.

Soln: Here,

$\varphi(2,3) = 0 \quad ; a < b$

Then,

$\varphi(4,3) = \varphi(4-3, 3) + 1$

$= \varphi(1,3) + 1$

$= 3 + 1$

$\therefore \varphi(4,3) = 4$

$$
\begin{aligned}
Q(11,3) &= Q(11-3,3)+1 \\
&= Q(8,3)+1 \\
&= 2+1 \\
\therefore Q(11,3) &= 3 \\
Q(8,3) &= Q(8-3,3)+1 \\
&= Q(5,3)+1 \\
\therefore Q(8,3) &= 1+1 \\
&= 2 \\
Q(5,3) &= Q(5-3,3)+1 \\
&= Q(2,3)+1 \\
&= 0+1 \\
\therefore Q(5,3) &= 1
\end{aligned}
$$

#Algorithm for insertion of an item in linear queue (Enqueue):

Step 1: Check if the queue is full.
Step 2: If the queue is full produce overflow error and exit.
Step 3: If the queue is not full increment rear pointer to point the next empty space.
Step 4: Add data element to the queue location where the rear is pointing.
Step 5: Return success.

OR

Step 1: Initialize front = -1 and rear = -1 (create an empty queue).
Step 2: If rear >= maxsize - 1 display queue overflow. and exit.
Step 3: Else set rear = rear + 1
Q[rear] = data;
Step 4: Stop

C funtion for enqueue operation:

```
#define N 5
int queue [N];
int front = -1;
int rear = -1;
void enqueue (int n)
{
    if (rear == N-1)
    {
        print (" Overflow Queue");
    }
    else if (front == -1 && rear == -1)
    {
        front = rear = 0;
        queue [rear] = n;
    }
}
```

```
else
    {
        rear++;
        queue[rear] = x;
    }
```

#Algorithm for deletion of an item (Dequeue):

Step 1: Check if the queue is empty.

Step 2: If the queue is empty produce underflow and exit.

Step 3: If the queue is not empty access the the data where front is pointing.

Step 4: Increment front pointer to point to the next available data element.

Step 5: Return success.

OR

Step 1: If front = -1 and rear = -1 display queue is empty (underflow) and exit.

Step 2: Else data = queue[front]
front = front + 1

Step 3: Stop

C function for enqueue operation:

```
void dequeue()
{
    if (front == -1 && rear == -1)
    {
        print("Queue is empty");
    }
else if (front == rear)
    {
        front = rear = -1;
    }
else
    {
        printf("%d", queue[front]);
        front++;
    }
}
```

Difference between Stack and Queue Data structures

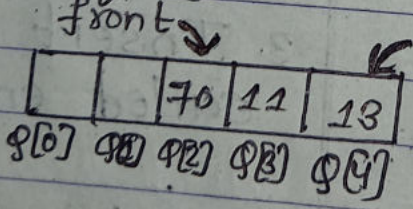
Stacks	Queues
1. Stacks are based on the LIFO principle, i.e., the element inserted at the last, is the first element to come out of the list.	1. Queues are based on the FIFO principle, i.e., the element inserted at the first, is the first element to come out of the list.
2. Insertion and deletion in stacks takes place only from one end of the list called the top.	2. Insertion & deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list & the deletion takes place from the front of the list.
3. Insert operation is called push operation.	3. Insert operation is called enqueue operation.
4. Delete operation is called pop operation.	4. Delete operation is called dequeue operation.

P.T.O. →

Stack	Queue
5. In stack, we maintain only one pointer to ~~across~~ access the list, called the top, which always points to the last element present in the list.	5. In queues, we maintain two pointers to access the list. The front pointer always points to the first element inserted in the list and is still present, and the rear pointer always points to the last inserted element.
6. Stack is used in sloving problems works on recursion.	6. Queue is used in solving problems having sequential processing.

Problem in linear queue:

Suppose a queue 'Q' has maximum size 5, say 5 elements ~~in~~ enqueue and two elements dequeue.



Now, if we attempt to add more elements, even though two queue cells are free, the elements cannot be inserted. Because in a queue, elements are always inserted at the rear end and

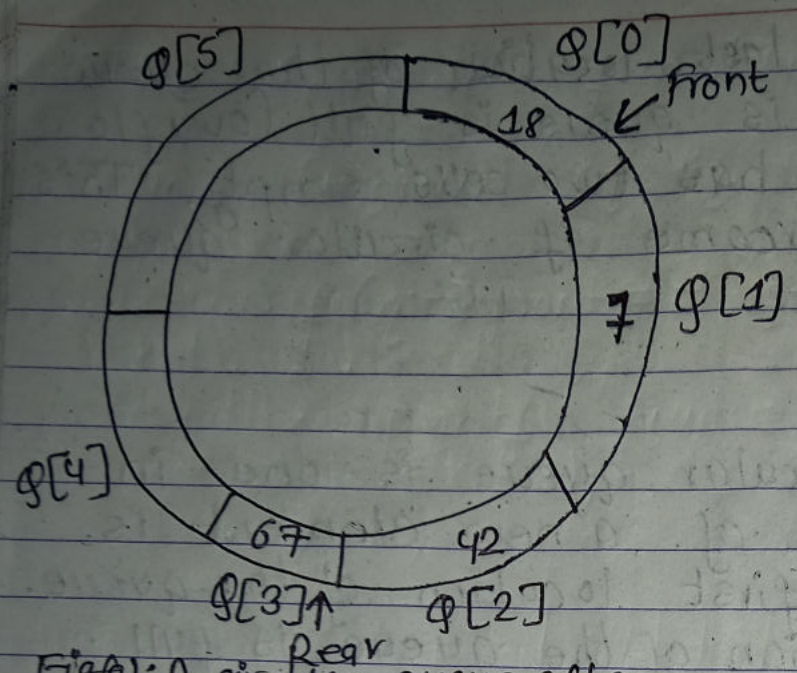
hence rear points to last location of the queue array Q[4]. That is queue is full (overflow condition) though it has two cells empty. This limitation can be overcome if circular queue is used.

#Circular Queue:

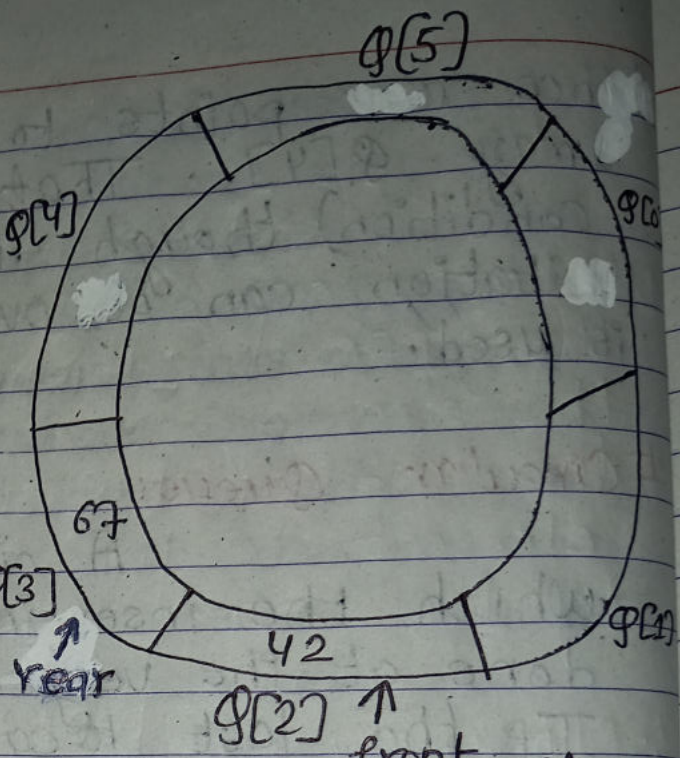
A circular queue is one in which the insertion of a new element is done at the very first location of the queue. If the last location of the queue is full. A circular queue overcomes the problem of unutilized space in linear queue.

In circular queue, we scarifies one element of the array thus to insert an element in a circular queue, we need an array of size n+1.

Suppose, Q is a queue array of six elements Enqueue() and Dequeue() operation can be performed on a circular queue. The following figures will illustrate the same.

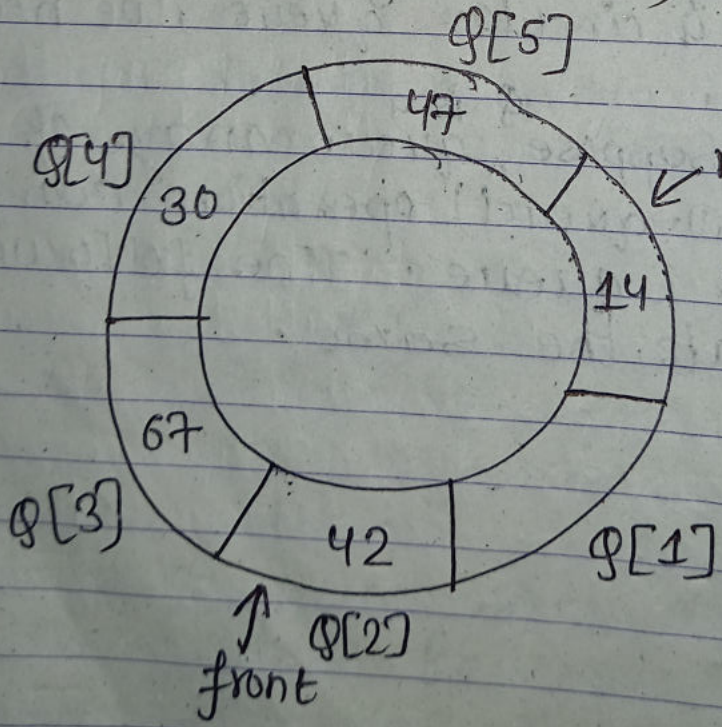


Fig(a): A circular queue after inserting 18, 7, 42 & 67.



Fig(b): Circular queue after deleting 18, 7

After inserting an element at last location Q[5], the next element will be inserted at the very first location (i.e. Q[0]).



Fig(c): Circular queue after inserting 30, 47, 14

At any time the relation will calculate the position of a element to be inserted by,

Rear = (rear+1) % maxsize

After deleting an element from circular queue, the position of front end is calculated by the relation,

front = (front+1) % maxsize

After locating the position of the new element to be inserted at rear compare it with front. If (rear == front) the queue has only one element.

Algorithm for insertion in a circular queue:

Step 1: Check queue full condition as
if (front == (rear+1) % maxsize)
print "queue is full" and exit.
else

Step 2: rear = (rear+1) % maxsize, (increment rear by 1)

Step 3: Cqueue[rear] = item;

Step 4: Stop

The Enqueue() function

```
void Enqueue (cq *q, int newitem)
{
  if (q->front == (q->rear+1) % maxsize)
  {
    printf ("queue is full");
    exit(1);
  }
  else
  {
    q->rear = (q->rear+1) % maxsize
    q->item [q->rear] = new item;
  }
}
```

#Algorithm for deleting an element in a circular queue:

~~Step 1: Check empty condition~~
~~if (rear == front)~~
~~print~~

Step 1: Check whether queue is Empty means check (front == -1)

Step 2: If it is empty then display Queue is empty.
If queue is not empty then go to step 3.

Step 3: Check if (front == rear) if it is true then set front = rear = -1
else

Step 4: front = (front + 1) % maxsize; (increment by 1)
item = cqueue[front];

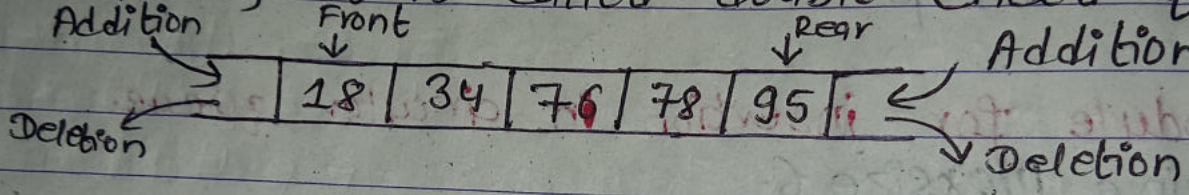
Step 5: Stop

#C module for insertion in circular queue.

```
#define Maxsize 6
int Q[maxsize], item;
void insert()
{
 int item;
 if (front == (rear+1) % maxsize)
 printf ("Queue overflow");
 else
 scanf ("value for item %d" &item);
 if (front==-1)
 {
    front=0;
    rear=0;
 }
 else
 {
   rear =(rear+1) % maxsize
    Q[rear]= item;
}}
```

DeQue (Double Ended Queue):

A DeQue is a homogeneous list in which elements can be added or inserted (enque operation) and deleted or removed from both the ends i.e. we can add a new element at the rear or front end and also can remove an element from both front and rear end. Hence, it is called double ended queue



fig(a): Deque

There are two types of DeQue depending upon the restriction to perform insertion or deletion operation at the two ends. They are:

(1) Input restricted ~~queue~~ deque.
(2) Output restricted ~~queue~~ deque.

(1) Input restricted ~~queue~~ deque:

An input restricted deque is a deque which allows insertion at only one end i.e. rear end but allows deletion from both ends i.e. rear and front end.

(2) Output restricted queue deque:

An output restricted deque is a deque which allows deletion at only one end i.e. front end but allows insertion at both ends i.e. front and rear end.

The possibe operation performed on deque is
(1) Add an element at the rear end.
(2) Add an element at the front end.
(3) Delete an element at the rear end.
(4) Delete an element at the front end.

Priority Queue:

Priority queue is a queue where each element is assigned a priority value. In priority queue, the elements are deleted and processed by following rules:

Rule 1: An element of higher priority is processed before any element of lower priority.

Rule 2: Two elements with the same priority are processed according to the order in which they were inserted to the queue.

Types of Priority:

(1) Ascending Priority Queue (Min Priority Queue)
(2) Descending " " (Max " ")

(1) Ascending Priority Queue:

It is the collection of item in which item can be inserted randomly but from which only the smallest item can be removed.

For example: We insert in order 8, 3, 2, 5 and they are removed in order 2, 3, 5, 8.

(2) Descending Priority Queue:

It is the collection of item in which item can be inserted randomly but from which only the largest item can be removed.

For example: We insert in order : 8, 3, 2, 5 and they are removed in order 8, 5, 3, 2.